• Python is case sensitive

• Python index starts from 0

• Python uses whitespace (tabs or spaces) to indent code instead of using braces. **HELP**

Help Home Page help() Function Help help(str.replace) Module Help help(re)

**MODULE (AKA LIBRARY)** Python module is simply a '.py' file

List Module Contents dir(module1) Load Module import module1 \* Call Function from Module module1.func1()

\* import statement creates a new namespace and executes all the statements in the associated *.py* file within that namespace. If you want to load the module's content into current namespace, use 'from module1 import \* '

**Check data type :** type(variable)

**SIX COMMONLY USED DATA TYPES** 1. **int/long\*** - Large int automatically converts to long 2. **float\*** - 64 bits, there is no 'double' type 3. **bool\*** - True or False 4. **str\*** - ASCII valued in Python 2.x and Unicode in Python 3

• String can be in single/double/triple quotes

• String is a sequence of characters, thus can be treated like other sequences

• Special character can be done via \ or preface with r

str1 = r'this\f?ff'

• String formatting can be done in a number of ways

template = '%.2f %s haha $%d'; str1 = template % (4.88, 'hola', 2)

Python Cheat Sheet

**JUST THE BASICS**

CREATED BY: ARIANNE COLTON AND SEAN CHEN

**SCALAR TYPES**

**GENERAL**

**SCALAR TYPES**

**\*** str(), bool(), int() and float() are also explicit type

cast functions.

5. **NoneType(None)** - Python 'null' value (ONLY one instance of None object exists)

• None is not a reserved keyword but rather a unique instance of 'NoneType'

• None is common default value for optional function arguments :

def func1(a, b, c = None)

• Common usage of None :

if variable is None :

6. **datetime** - built-in python 'datetime' module provides 'datetime', 'date', 'time' types.

• 'datetime' combines information stored in 'date' and 'time'

Create datetime from String

dt1 = datetime. strptime('20091031', '%Y%m%d')

Get 'date' object dt1.date()

Get 'time' object dt1.time() Format datetime to String

**Note :** All non-Get function call i.e. list1.sort() examples below are in-place (without creating a new object) operations unless noted otherwise.

**TUPLE**

One dimensional, fixed-length, **immutable** sequence of Python objects of ANY type.

dt1.strftime('%m/%d/%Y %H:%M')

Change Field dt2 = dt1.replace(minute = Value 0, second = 30) Get Difference diff = dt1 - dt2

# diff is a 'datetime.timedelta' object

**Note :** Most objects in Python are mutable except for 'strings' and 'tuples'

**DATA STRUCTURES**

• **DATA STRUCTURES**

Create Tuple tup1 = 4, 5, 6 or tup1 = (6,7,8) Create Nested Tuple tup1 = (4,5,6), (7,8) Convert Sequence Iterator to Tuple or tuple([1, 0, 2])

**Note :**

• 'start' index is included, but 'stop' index is NOT.

• start/stop can be omitted in which they default to the start/end.

Concatenate Tuples tup1 + tup2 Unpack Tuple a, b, c = tup1

**§** Application of 'step' :

**Application of Tuple**

Take every other element list1[::2]

Swap variables b, a = a, b

Reverse a string str1[::-1]

**LIST**

**DICT (HASH MAP)** One dimensional, variable length, **mutable** (i.e. contents can be modified) sequence of Python objects of ANY type.

Create Dict dict1 = {'key1' : 'value1' , 2 :[3, 2]} Create Dict from Create List list1 = [1, 'a', 3] or list1 = list(tup1)

Sequence

Concatenate Lists\* list1 + list2 or list1.extend(list2) Append to End of List list1.append('b') Insert to Specific Position

dict(zip(keyList, valueList)) Get/Set/Insert Element dict1['key1']\* dict1['key1'] = 'newValue' Get with Default Value dict1.get('key1', defaultValue) \*\* list1.insert(posIdx, Check if Key Exists 'key1' in dict1 'b') \*\*

Delete Element del dict1['key1'] Inverse of Insert valueAtIdx = list1. pop(posIdx) Remove First Value from List list1.remove('a') Check Membership 3 in list1 => True \*\*\*

Get Key List dict1.keys() \*\*\* Get Value List dict1.values() \*\*\* Update Values dict1.update(dict2)

# dict1 values are replaced by dict2 Sort List list1.sort() Sort with User- \* 'KeyError' exception if the key does not exist. Supplied Function

\*\* 'get()' by default (aka no 'defaultValue') will return 'None' if the key does not exist.

\*\*\* Returns the lists of keys and values in the same order. However, the order is not any particular order, aka it is most likely not sorted.

**Valid dict key types**

• Keys have to be immutable like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable too)

• The technical term here is 'hashability', check whether an object is hashable with the hash('this is string'), hash([1, 2]) - this would fail. **SET**• A set is an **unordered** collection of UNIQUE elements.

• You can think of them like dicts but keys only.

Create Set set([3, 6, 3]) or {3, 6, 3} Test Subset set1.issubset (set2) Test Superset set1.issuperset (set2) Test content sets have same set1 == set2

• **Set operations :**

Union(aka 'or') set1 | set2 Intersection (aka 'and') set1 & set2 Difference set1 - set2 Symmetric Difference (aka 'xor') set1 ^ set2 list1.sort(key = len) # sort by length

\* List concatenation using '+' is expensive since a new list must be created and objects copied over. Thus, extend() is preferable.

\*\* Insert is computationally expensive compared with append.

\*\*\* Checking that a list contains a value is lot slower than dicts and sets as Python makes a linear scan where others (based on hash tables) in constant time.

**Built-in 'bisect module**‡

• Implements binary search and insertion into a sorted list

• 'bisect.bisect' finds the location, where 'bisect. insort' actually inserts into that location.

**‡** WARNING : bisect module functions do not check whether the list is sorted, doing so would be computationally expensive. Thus, using them in an unsorted list will succeed without error but may lead to incorrect results.

**SLICING FOR SEQUENCE TYPES**†

† Sequence types include 'str', 'array', 'tuple', 'list', etc.

Notation list1[start:stop]

list1[start:stop:step] (If step is used) §

1. **FUNCTIONS** Python is **pass by reference**, function arguments

**OBJECT-ORIENTED PROGRAMMING**

**EXCEPTION HANDLING**

1. Basic Form : are passed by reference.

• Basic Form :

def func1(posArg1, keywordArg1 = 1, ..):

**'object'** is the root of all Python types 2. Everything (number, string, function, class, module, etc.) is an object, each object has a 'type'. Object try: .. except ValueError as e: variable is a pointer to its location in memory. 3. All objects are reference-counted.

sys.getrefcount(5) => x

a = 5, b = a # This creates a 'reference' to the object on the right side of =, thus both a and b point to 5

sys.getrefcount(5) => x + 2

del(a); sys.getrefcount(5) => x + 1

4. **Class** Basic Form :

class MyObject(object):

# 'self' is equivalent of 'this' in Java/C++ def \_\_init\_\_(self, name):

self.name = name def memberFunc1(self, arg1):

.. @staticmethod def classFunc2(arg1):

.. obj1 = MyObject('name1') obj1.memberFunc1('a') MyObject.classFunc2('b')

5. Useful interactive tool :

dir(variable1) # list all methods available on the object**COMMON STRING OPERATIONS**

Concatenate List/Tuple with Separator print e except (TypeError, AnotherError): .. except: **Note :** • Keyword arguments MUST follow positional arguments.

.. finally:

.. # clean up, e.g. close db

• Python by default is NOT "lazy evaluation", 2. Raise Exception Manually expressions are evaluated immediately.

raise AssertionError # assertion failed

• Function Call Mechanism :

raise SystemExit # request program exit raise RuntimeError('Error message : 1. All functions are local to the module level

..') scope. See 'Module' section. 2. Internally, arguments are packed into a tuple

and dict, function receives a tuple 'args' and dict 'kwargs' and internally unpack.

• Common usage of 'Functions are objects' :

def func1(ops = [str.strip, user\_ define\_func, ..], ..): for function in ops: value = function(value)

**RETURN VALUES**

• **None** is returned if end of function is reached without encountering a return statement.

• Multiple values return via ONE tuple object

return (value1, value2) value1, value2 = func1(..)

**ANONYMOUS (AKA LAMBDA) FUNCTIONS**

• What is Anonymous function? A simple function consisting of a single statement.

lambda x : x \* 2 # def func1(x) : return x \* 2

• Application of lambda functions : 'curring' aka deriving new functions from existing ones by partial argument application.

ma60 = lambda x : pd.rolling\_mean(x, 60)

**USEFUL FUNCTIONS (FOR DATA STRUCTURES)**

1. **Enumerate** returns a sequence (i, value) tuples

where i is the index of current item.

for i, value in enumerate(collection):

• Application : Create a dict mapping of value of a sequence (assumed to be unique) to their locations in the sequence.

• Application :

sorted(set('abc bcd')) => [' ', 'a', 'b', 'c', 'd'] # returns sorted unique characters

3. **Zip** pairs up elements of a number of lists, tuples or other sequences to create a list of tuples :

zip(seq1, seq2) => [('seq1\_1', 'seq2\_1'), (..), ..]

• Zip can take arbitrary number of sequences. However, the number of elements it produces is determined by the 'shortest' sequence.

• Application : Simultaneously iterating over multiple sequences :

for i, (a, b) in enumerate(zip(seq1, seq2)):

**LIST, SET AND DICT**

• Unzip - another way to think about this is converting a list of rows to a list of columns.

**COMPREHANSIONS**

Syntactic sugar that makes code easier to read and write seq1, seq2 = zip(\*zipOutput)

1. **List comprehensions**

4. **Reversed** iterates over the elements of a sequence

in reverse order.

• Concisely form a new list by filtering the elements of a collection and transforming the elements passing the filter in one concise expression.

list(reversed(range(10))) \*

• Basic form :

[expr for val in collection if condition] **\*** reversed() returns the iterator, list() makes it a list.

A shortcut for :

result = [] **CONTROL AND FLOW**

for val in collection:

if condition:

result.append(expr)

1. Operators for conditions in 'if else' :

The filter condition can be omitted, leaving only the expression. Check if two variables are same object var1 is var2 . . . are different object var1 is not var2 Check if two variables have same value var1 == var2

2. **Dict Comprehension**

• Basic form :

{key-expr : value-expr for value in collection if condition}

**WARNING :** Use 'and', 'or', 'not' operators for compound conditions, not &&, ||, !.

', '.join([ 'v1', 'v2', 'v3']) => 'v1, v2, v3'

string1 = 'My name is {0} {name}'

2. Common usage of 'for' operator :

Format String newString1 = string1. format('Sean', name = Iterating over a collection (i.e. list or tuple) or an iterator

for element in 'Chen') iterator :

3. **Set Comprehension**

• Basic form : same as List Comprehension except with curly braces instead of []

4. **Nested list Comprehensions**

sep = '-';

• Basic form :

. . . If elements are sequences, can be 'unpack'

for a, b, c in iterator :

Split String

stringList1 = string1.split(sep)

[expr for val in collection for innerVal in val if condition] 3. 'pass' - no-op statement. Used in blocks where no action is to be taken. 4. Ternary Expression - aka less verbose 'if else'

Get Substring start = 1; string1[start:8]

• Basic Form :

Created by Arianne Colton and Sean Chen data.scientist.info@gmail.com

2. **Sorted** returns a new sorted list from any sequence

value = true-expr if condition sorted([2, 1, 3]) => [1, 2, 3]

else false-expr

String Padding with Zeros

month = '5'; month.zfill(2) => '05'

Based on content from month = '12';

'Python for Data Analysis' by Wes McKinney 5. No switch/case statement, use if/elif instead.

month.zfill(2) => '12'

Updated: May 3, 2016